VFS 2ème partie

Rappels

- Le VFS est une couche d'abstraction
- Masque les détails d'implémentation du système de fichiers
- Nous avons vu
  - Comment créer un périphérique virtuel (mknode)
  - Comment écrire un driver pour ce périphérique
  - Utilisation des major/minor
  - Mais limité à un fichier (open, read, write…)
- Approche objet
  - Implémentation avec des structures
  - Les opérations spécifiques sont fournies à travers des pointeurs sur fonctions (file_operations pour les fichiers)

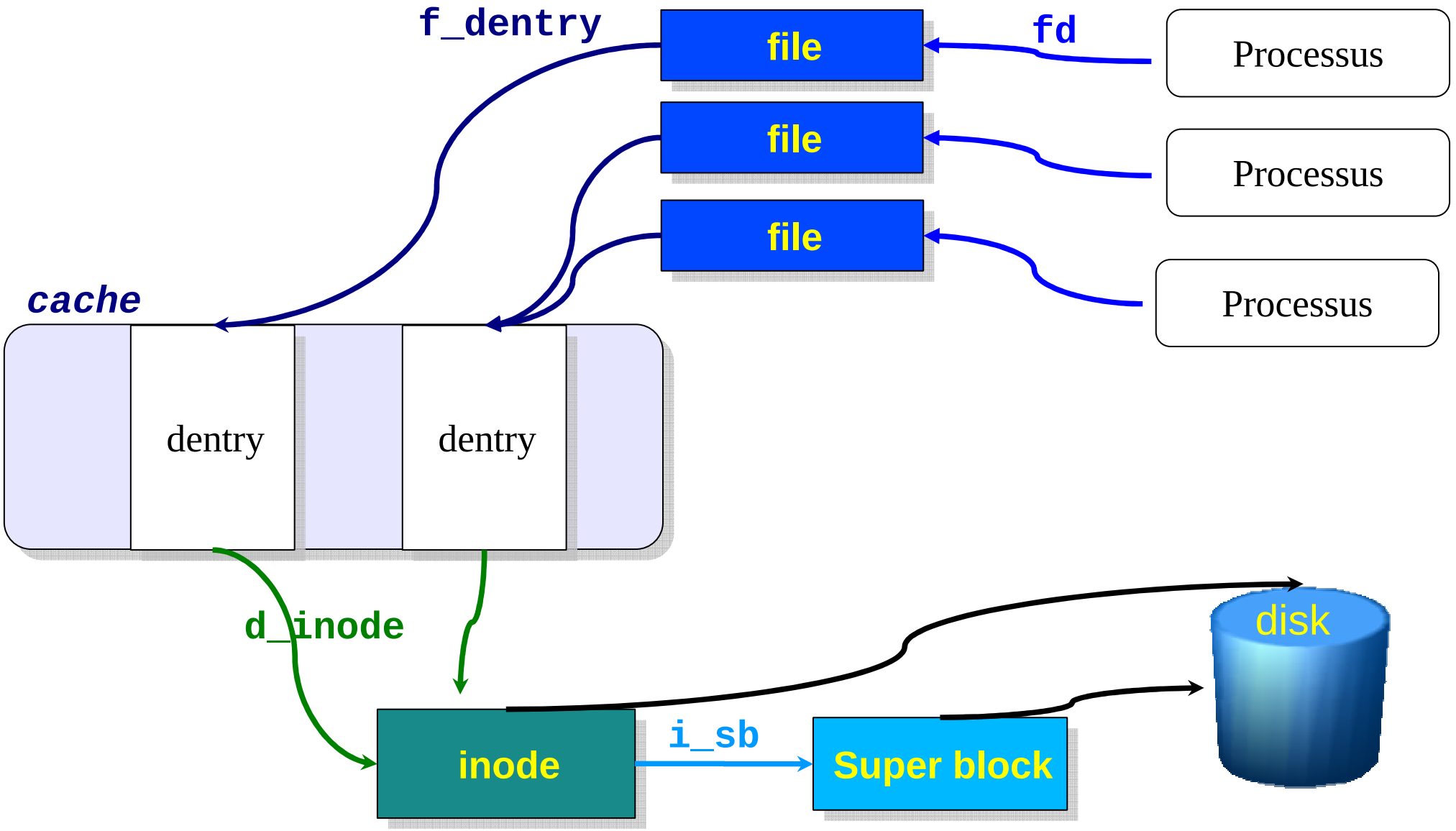
Les Principaux Objets de VFS

- File : Représente tout ce sur quoi on peut lire/écrire
  - Informations sur un fichier ouvert par un processus
  - N'existe que dans la mémoire du kernel
  - Fournit l'implémentation des opérations disponibles sur le fichier
  - Définie dans `<linux/fs.h>`
- Inode : Représente un objet dans le système de fichiers
  - Peut être un fichier, un répertoire, un lien symbolique…
  - Chaque Inode a un numéro unique (*inode number*) qui permet d'identifier l'objet correspondant
  - Certains objets ont un inode mais pas de structure file (liens symboliques) et vice-versa
  - Définie dans `<linux/fs.h>`

Les Principaux Objets de VFS

- Dentry : informations sur le lien entre une entrée de répertoire et le fichier correspondant
  - On ne veut pas manipuler des inodes mais des noms
  - L'association nom-inode peut être coûteuse
  - VFS maintient une liste de nom actifs ou récemment utilisés dans un arbre, le *dcache*
  - Les nœuds de cet arbre sont des *dentries*
  - Les *dentries* sont des intermédiaires entre les *files* et les *inodes*
  - Définie dans `<linux/dcache.h>`
- Super Block : Informations globales sur le FS
  - État courant de la partition
  - Définie dans `<linux/fs.h>`

Rappels


f_dentry
file
fd
Processus
file
Processus
file
Processus
cache
dentry
dentry
d_inode
disk
inode
i_sb
Super block

VFS et systèmes de fichiers

- Opérations pour utiliser un vrai système de fichiers
- Être capable de le monter
  - Structure *superblock*
- Trouver des fichiers
  - Conversion des chemins (*Pathname lookup*)
- Gérer les inodes
- Un système de fichier est décrit par un superblock
  - *struct super_block*
  - Peut exister physiquement dans le système de fichier (ext2) ou être crée dynamiquement (FAT)

La Structure du Super Block

struct list_head	s_list
kdev_t	s_dev
unsigned long	s_blocksize
unisgned char	s_blocksize_bits
unsigned char	s_dirt
unsigned long long	s_maxbytes
struct file_system_type *	s_type
struct super_operations *	s_op
struct dquot_operations *	dq_op
unsigned long	s_flags
unisgned long	s_magic
struct dentry *	s_root
struct rw_semaphore	s_umount
struct semaphore	s_lock
int	s_count
atomic_t	s_active
struct list_head	s_dirty
struct list_head	s_locked_inodes
struct list_head	s_files
struct block_device *	s_bdev
struct list_head	s_instances
struct quot_mount_options	s_dquot
union	u

La Structure du Super Block

- *struct list_head s_list*
  - Liste chaînée des *super blocks* actuellement dans le système
- *unsigned char s_dirt*
  - Le *superblock* existe toujours en mémoire
  - Il peut aussi exister sur le disque
  - Problème de synchronisation
- *struct file_system_type * s_type*
  - Donne des informations sur le système de fichiers
- *unsigned long magic*
  - Identifiant unique du système de fichiers
- *struct dentry * root*
  - Racine du système de fichiers

La Structure du Super Block

- *struct list_head dirty*
  - Liste d'inodes *dirty*
- *struct list_head files*
  - Liste des fichiers ouverts
- *struct super_operations * s_op*
  - Pointeur vers la structure *super_operations*
  - Contient les fonctions utilisables sur ce sb
- *struct dquot_operations * d_op*
  - Pointeur vers structure *dquot_operations*
  - Opérations sur les quotas
- *union u*
  - Contient les informations spécifiques au FS
  - Plusieurs spécifiées (*struct minix_sb_info, struct ext2_sb_info...*)
  - Extensible (*void * generic_sbp*)

struct super_operations

- read_inode(inode)
- dirty_inode(inode)
- write_inode(inode,flag)
- put_inode(inode)
- delete_inode(inode)
- put_super(super)
- write_super(super)
- unlockfs(super)
- statfs(super,buf)
- remount_fs(super,flags,data)
- clear_inode(inode)
- umount_begin(super)
- show_options(seq_file,vfsmount)

Super block operations

- *read_inode(struct inode *)*
  - Appelée pour lire un inode spécifique depuis le FS
  - Les champs *i_sb*,*i_dev* et *i_no* sont fixés pour indiquer quel inode doit être lu sur quel FS
  - Doit donner une valeur aux *inode_operations*
- *dirty_inode(struct inode *)*
  - Marque l'inode comme *dirty*, c'est-à-dire différent en mémoire et sur disque
- *write_inode(struct inode *,flag)*
  - Écrit l'inode sur disque
  - Appelé sur les inodes *dirty*
  - Le *flag* indique si l'écriture doit être synchrone ou pas (utilisation suivant FS)
- *put_inode(struct inode *)*
  - Si défini, appelé à chaque fois que le compteur d'utilisation d'un inode est diminué
  - Appelé avant la décrémentation
  - Utilisé pour traitement spécifique quand dernière référence fermée

Super block operations

- *delete_inode(struct inode *)*
  - Si elle existe, appelée quand le compteur d'utilisation atteint 0 et quand le compteur de liens (*i_nlink*) vaut aussi 0
- *write_super(struct super_block *)*
  - Appelé pour écrire le sb sur le disque
- *put_super(struct super_block *)*
  - Appelé à la fin d'un umount
  - Utilisé typiquement pour enlever des structures spécifiques au FS (bitmap blocs libres…)
- *statfs(struct super_block *, struct kstatfs *)*
  - Fournit des statistiques sur le FS
- *show_options(struct seq_file *,struct vfsmount *)*
  - Informations affichées dans */proc/<pid>/mounts*

La Structure de l'inode

struct list_head	i_hash
struct list_head	i_list
struct list_head	i_dentry
unsigned long	i_ino
unsigned int	i_count
kdev_t	i_dev
umode_t	i_mode
nlink_t	i_nlink
uid_t	i_uid
gid_t	i_gid
kdev_t	r_dev
time_t	i_atime
time_t	i_mtime
time_t	i_ctime
unsigned long	i_blksize
unsigned long	i_blocks
unsigned long	i_version
unsigned long	i_nrpages
struct semaphore	i_sem
struct semaphore	i_atomic_write
struct inode_operations *	i_op
struct file_operations *	i_fop
struct super_block *	i_sb

La Structure de l'inode

struct file_lock *	i_flock
struct vm_area_struct *	i_list
struct list_head	i_mmap
struct page *	i_pages
struct dquot **	i_dquot
unsigned long	i_state
unsigned int	i_flags
unsigned char	i_pipe
unsigned char	i_sock
int	i_writecount
unsigned int	i_attr_flags
__u32	i_generation
union	u
struct wait_queue *	i_wait

La Structure de l'inode

- *struct list_head i_hash;*
  - Relie tous les inodes se trouvant dans le même *hash bucket*
  - La valeur de hachage est calculée à partir de l'adresse de la structure *super_block* et le numéro d'inode
- *struct list_head i_list;*
  - Sert à construire 3 listes pour les inodes
  - *inode_in_use*
  - *inode_unused*
  - *superblock->s_dirty*
- *struct list_head i_dentry*
  - Liste de toutes les *struct dentry* qui référencent cet inode
- *unsigned long i_ino*
  - Numéro d'inode
- *unsigned int i_count*
  - Compteur de références

La Structure de l'inode

- *unsigned long i_state*
  - 3 états possibles
  - I_DIRTY
    - Membre de la liste *s_dirty* du *superblock*
    - Sera écrit sur disque lors du prochain sync
  - I_LOCK
    - Les inodes sont verrouillés lorsqu'ils sont lus/écris/crées
  - I_FREEING
    - Compteur de référence et de lien à 0
- *time_t [acm]time*
  - Heure de création, modification, accès
- *u32 i_generation*
  - Numéro de génération de cet inode
  - Utilisé par nfs et ext2

struct inode_operations

- struct file_operations * default_file_ops
- int create(struct inode *, struct dentry *)
- struct dentry * lookup (struct inode *, struct dentry *)
- int link (struct dentry *,struct inode *, struct dentry *)
- int unlink (struct inode *, struct dentry *)
- int symlink (struct inode *, struct dentry *, const char *)
- int mkdir (struct inode *, struct dentry *,int)
- int rmdir (struct inode *, struct dentry *)
- int mknod (struct inode *, struct dentry *,int,int)
- int rename (struct inode *, struct dentry *, struct inode *, struct dentry *)
- int readlink (struct inode *, char *, int)
- struct dentry * follow_link (struct dentry *, struct dentry *, unsigned int)
- int get_block (struct inode *, long, struct buffer_head *, int)
- int readpage(struct file *, struct page *)
- int writepage(struct file *, struct page *)
- int flushpage(struct inode *, struct page *, unsigned long)
- void truncate (struct inode *)
- int permission(struct inode *, int)
- int smap(struct inode *, int)
- int revalidate(struct dentry *)

struct inode_operations

- default_file_ops
  - Pointe sur les file_operations du fichier correspondant à cet inode
  - Quand un fichier est ouvert, le champs f_op est initialisé à partir de celui-ci, puis la méthode *open* est appelée
  - Mais plus utilisé (champs file_operations dans inode)
- create
  - Appelé par VFS pour créer un fichier dont le nom et le répertoire sont dans la dentry
  - VFS aura déjà vérifié qu'il n'y a pas de fichier de même nom existant
  - Devra
    - Prendre un nouvel inode dans le cache avec *get_empty_inode* et le remplir
    - L'insérer dans la table de hachage avec *insert_inode_hash*
    - Le marquer *dirty*
    - L'instancier dans le *dcache*

struct inode_operations

- lookup
  - Vérifie que le nom donné dans le *dentry* existe dans le répertoire donné par l'inode
  - Retourne un *dentry* negatif en cas d'erreur
  - Ajout de la dentry modifiée au cache si réussite avec *d_add()*
- link
  - Création d'un lien *hard* du nom donné dans le premier *dentry* vers le second, dans le repertoire indiqué par *inode*
- mkdir
  - Crée un répertoire dont le parent est l'*inode*, le nom est la *dentry* et avec le mode indiqué

struct inode_operations

- rename
  - Renomme le *dentry* du répertoire *inode*
  - Les vérifications courantes (existence, nouveau nom pas fils de l'ancien…) ont déjà été effectuées
- readpage/writepage/flushpage
  - Utilisé par les fichiers mappés en mémoire
- truncate
  - Modifie la taille du fichier associé à l'inode
  - Avant d'appeler cette méthode, il faut modifier *i_size*
- revalidate
  - Met à jour les attributs en cache du fichier spécifié par la dentry
  - Utilisé par NFS

Objets dentry

- Dans VFS, un répertoire est un fichier
  - Il contient une liste de fichiers et de répertoires
- Quand un répertoire est lu, il est transformé en une série d'objets *dentry*
  - Un objet par entrée du répertoire
- Ces objets n'ont aucune existence sur le disque
  - Création coûteuse
- Un chemin vers un fichier amenera à la création de plusieurs dentries
  - /tmp/toto nécessitera 3 dentries
- Les dentries sont maintenues dans un cache

État d'une dentry

- Chaque dentry peut être dans un état parmi 4
- *Free*
  - La dentry ne contient pas d'information valide et n'est pas utilisée par VFS
- *Unused*
  - Ce dentry n'est actuellement pas utilisé
  - Son *d_count* est à 0 mais son *d_inode* pointe encore vers l'inode associé
  - Il contient encore des informations valides mais peut être supprimé pour récupérer de la mémoire
- *In Use*
  - Actuellement utilisé
  - Son *d_count* est positif et son *d_inode* pointe vers l'inode associé
  - Ne peut être supprimé de la mémoire
- *Negative*
  - L'inode associé n'existe plus parce que celui sur disque a été effacé
  - Gardé en mémoire pour résolution rapide

La Structure d'un dentry

int	d_count
unsigned int	d_flags
struct inode *	d_inode
struct dentry *	d_parents
struct dentry *	d_mounts
struct dentry *	d_covers
struct list_head	d_hash
struct list_head	d_lru
struct list_head	d_child
struct list_head	d_subdirs
struct list_head	d_alias
struct qstr	d_name
unsigned long	d_time
struct dentry_operations *	d_op
struct super_block *	d_sb
unsigned long	d_reftime
void *	d_fsdata
unsigned char	d_iname[16]

Structure d'un dentry

- *d_count*
  - Compteur de référence
  - N'inclus pas les références des parents, mais des enfants
- *d_inode*
  - Pointeur vers l'inode correspondant à ce nom
  - Peut être NULL dans le cas d'un dentry négatif
- *d_parent*
  - dentry parent
  - Pour la racine, peut être un pointeur sur soi
- *d_mounts*
  - Si le répertoire est le point de montage d'un FS, pointe vers la racine de celui-ci
- *d_covers*
  - Symétrique de *d_mounts*

Structure d'un dentry

- *d_child*
  - Autres enfants du parent (i.e. les frères!)
- *d_subdirs*
  - Liste de tous les fils de la dentry
  - Peuvent être des fichiers!
- *d_alias*
  - A cause des liens hards, un fichiers peut avoir plusieurs noms
  - Cette liste maintient ensemble les dentries qui concernent le même fichier
- *d_name*
  - Contient le nom de l'entrée et sa valeur de hachage
- d_iname
  - Contient les 16 premiers caractères du nom pour accès rapide
  - Si nom complet possible, alors d_name.name pointe dessus, sinon pointe sur une zone mémoire

Le cache dentry

- Il contient
  - Un ensemble de dentries dans un des 3 états possibles (utilisés, non utilisé, négatif)
  - Une table de hachage qui permet de trouver le dentry associé à un nom de fichier et de répertoire
    - Résolution des collisions par chaînage
    - Taille dépendante de la RAM
- Les dentries *Unused* et *Negative* sont maintenues dans une liste
  - *d_lru* , triée par date d'insertion
  - Si mémoire nécessaire, le kernel peut enlever les éléments en fin de chaîne

dentry_operations

- *int d_revalidate (struct dentry *, int)*
  - Appelé quand un lookup de chemin utilises une entrée dans le cache, pour vérifier sa validité
  - Utilie seulement si le FS peut changer sans intervention de VFS
- *int d_hash(struct dentry *, struct qstr *)*
  - Calcule la valeur de hachage (si non standard)
  - Le *dentry* est le **parent**
- *int d_compare(struct dentry *, struct qstr *, struct qstr*)*
  - Compare les 2 *qstr*
  - dentry doit être leur parent

dentry_operations

- *void d_delete(struct dentry *)*
  - Appelé quand le compteur de référence atteint 0
  - Avant d'être placé dans la lru
- *void d_release(struct dentry *)*
  - Appelé avant de libérer la mémoire d'une dentry
  - Permet de faire le ménage (d_fsdata)
- *void d_iput(struct dentry *, struct inode *)*
  - Si défini, est appelé à la place d'iput pour libérer la mémoire d'un inode quand la dentry est supprimée

Ajout d'un FS

- Bon, comment on fait en pratique?
- On commence par déclarer au kernel que le module gère un nouveau FS
- Utilisation de la structure *file_system_type* de *<include/linux/fs.h>*

```
struct file_system_type {
        const char *name;
        int fs_flags;
        struct super_block *(*read_super) (struct super_block *, void *, int);
        struct file_system_type * next;
};
```

- Exemple pour ext2fs

```
static struct file_system_type ext2_fs_type = {
        "ext2", FS_REQUIRES_DEV , ext2_read_super, NULL };
```

- Et on l'enregistre avec *int register_filesystem*

struct file_system_type

- *name*
  - Nom du FS tel que visible dans */proc/filesystems*
  - Doit être unique
- *fs_flags*
  - *FS_REQUIRE_DEV* : le FS ne peut être monté que sur un block device
  - *FS_SINGLE* : Ne peut avoir qu'un superblock et doit être monté par *kern_mount*
  - *FS_NOMOUNT* : ne peut être mount depuis le user space
- *read_super*
  - Pointeur vers la fonction qui lit le superblock pendant le montage
  - Si non fourni, echec du montage (et OOPS kernel parfois)
- *owner*
  - THIS_MODULE
- *kern_mnt*
  - Utilisé lors du FS_SINGLE
- *next*
  - Liste chainée de FS

Rôle de read_super

- Le role de read_super est de
  - Remplir les champs du superblock
  - Allouer l'inode racine
  - Initialiser les données propres du FS
- Implémentation
  - Lire le superblock sur le périphérique (*sb->s_dev*)
  - Vérifier que le superblock contient le *magic number* et a l'air ok
  - Initialise le *sb->s_op* pour pointer vers les sb operations
  - Alloue l'inode racine avec *new_inode* et la dentry racine avec *d_alloc_root*
  - Si le FS n'est pas monté read-only, marquer le sb comme dirty

Montage d'un FS

- Quand un FS est monté, appel système à *do_umount()*
- *do_umount* appelle *read_super*
  - Utilisation de la méthode indiquée dans la struct super_operations
- Retourne une struct superblock qui contient le pointeurs vers les fonctions (*super_operations*)
- C'est tout ☺

Trouver un fichier

- 2 étapes
  - Trouver l'inode correspondant au nom
  - Accéder à l'inode
- Le chemin vers le fichier est découpé en éléments
- Traités séquentiellement
  - Si élément à chercher est un répertoire, alors le suivant est cherché relativement à lui
  - Chaque élément recherché retourne un *inode*
  - Si lien symbolique, la recherche reprend avec le nouveau chemin
    - Limite au nombre max de symlinks
- Mais comment commencer la recherche?
  - Il nous faut un inode
  - Celui de la racine!

Trouver un fichier

- Récupération de l'inode de la racine
  - Champs *s_root*
- Utilisation des inode operations
  - Méthode *lookup()*
- Exemple, recherche de */toto/titi*
  - Récupération de l'inode « / »
  - Appel de lookup sur cet inode avec une dentry *toto*
  - Etc

Parcourir un répertoire

- Comment obtenir la liste des fichiers/répertoires contenus dans un répertoire?
- Déjà obtenir l'inode/dentry du répertoire
  - Facile avec lookup()
- Ensuite lister son contenu…
- Un répertoire, c'est en fait un fichier
  - Donc utilisation des file_operations!
  - Méthode readdir()
- *readdir(struct file *filp,void *dirent,filldir_t filldir)*
  - 3 paramètres
    - Pointeur vers répertoire
    - Pointeur où mettre le résultat
    - Méthode à utiliser pour créer une entrée de répertoire (fournie par VFS)

Parcourir un répertoire

- Principe de fonctionnement de *readdir*
- Étant donné le filp, trouver la dentry correspondante
- Avec la dentry, on peut trouver son inode
- Le flip->pos indique l'offset, c'est-à-dire le numéro d'entrée dans le répertoire
  - 0 : "."
  - 1 : ".."
- Grâce à l'offset, on peut garder une trace des readdir successifs
  - Listing d'un répertoire!
- Si l'entrée existe, alors utilisation de la méthode filldir

Parcourir un répertoire

- filldir(void * __buf, const char *name, int namelen, loff_t offset, inot_ino, unsigned int d_type)
  - Buffer où mettre l'entrée
  - Nom de l'entrée
  - Longueur du nom
  - Numéro de l'entrée
  - Inode de l'entrée
  - Type d'entrée (pour optimisation, DT_DIR pour un répertoire)
- Exemple: entrée « . »
  - filldir(dirent, « . », 1, 0, inodeQuiVaBien, DT_DIR)